IMPLEMENTATION OF MOTION ESTIMATION BASED ON HETEROGENEOUS PARALLEL COMPUTING SYSTEM WITH OPENC

Jinglin Zhang, Jean François Nezan, Jean-Gabriel Cousin

IMPLEMENTATION OF MOTION ESTIMATION BASED ON HETEROGENEOUS PARALLEL COMPUTING SYSTEM WITH OPENCL

Jinglin ZHANG, Jean-Francois NEZAN, Jean-Gabriel COUSIN.

Université Européenne de Bretagne, France
INSA, IETR, UMR CNRS 6164
20, Avenue des Buttes de Coesmes, 35708 RENNES , France
Email:jinglin.zhang,jnezan,jcousin@insa-rennes.fr

ABSTRACT

Heterogeneous computing system increases the performance of parallel computing in many domain of general purpose computing with CPU, GPU and other accelerators. Open Computing Language (OpenCL) is the first open, royalty-free standard for heterogenous computing on multi hardware platforms. In this paper, we propose a parallel Motion Estimation (ME) algorithm implemented using OpenCL and present several optimization strategies applied in our OpenCL implementation of the motion estimation. In the same time, we implement the proposed algorithm on our heterogeneous computing system which contains one CPU and one GPU, and propose one method to determine the balance to distribute the workload in heterogeneous computing system with OpenCL. According to experiments, our motion estimator with achieves 100× to 150× speed-up compared with its implementation with C code executed by single CPU core and our proposed method obtains obviously enhancement of performance in based on our heterogeneous computing system.

Index Terms— Motion Estimation, OpenCL, Heterogeneous, Parallel, CPU, GPU

I. INTRODUCTION

As a result of continued demand of Video Quality and Compression Rate, the efficiency and complexity of Video Coding standards face with some great challenges. The ME is still the main time consuming task of a video encoder with 40% to 60% of the computation load. The goal of the ME is to find relative motion between two images in order to eliminate temporal redundancy. For video compression, block matching algorithms are most widely used. An example is the full search where every candidate within a search window of magnitude p pixels is considered. The full search ME is very computationally intensive so that some fast algorithms like EPZS [**?**] had been proposed to accelerate the ME on a single processor core. The number of evaluated candidates is decreased but it comes with a loss in terms of quality (lower PSNR of the compressed video).

Urban in [**?**] proposed one real-time ME for H.264 high definition video encoding on multi-core DSP, well-suited for embedded parallel systems. New heterogeneous embedded systems like Tegra2 [**?**] can integrate low power GPU, CPU (ARM cores) and some algorithms like ME can take benefits of this new feature [**?**]. Some first results had been proposed [**?**], [**?**] using the CUDA approach. In this context, the parallel structure of the full search is used to accelerate the computation without decreasing the PSNR.

General-Purpose computing on Graphics Processing Units (GPGPU) is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. The OpenCL is a framework executing programs on heterogeneous platforms consisting of CPUs, GPUs and other dedicated processors. OpenCL is a good candidate in comparison with the CUDA approach specifically developed for GPU platforms from Nvidia, especially for the future heterogeneous embedded systems.

In this paper, we propose a parallelized full search ME algorithm and apply proposed algorithm on heterogeneous computing system with OpenCL. Section 2 presents the optimization strategies used to achieve a high-performance ME implementation. Section 3 illustrates our approach of motion estimation with OpenCL. Section 4 applies our proposed algorithm on the heterogeneous computing system with OpenCL and present the method to distribute workload in heterogenous system. A brief conclusion is given in section 5.

II. OPTIMIZATION STRATEGIES OF OPENCL

In this section, we present some optimization strategies used in our work. Table.**??** lists the relationship between the concepts of devices and those of OpenCL.

II-A. Fully saturate the computing resource

Using OpenCL, programmers should think about the practical infrastructure of different platforms. Although OpenCL has a unified Programming Model, it cannot ensure that the

Table I. Relationship between Devices and OpenCL

Devices	OpenCL
thread	work-item
threads group	work-group
shared memory	local memory
device memory	global memory

same OpenCL kernel has the best performance for different platforms. For instance, NVIDIA's GPU support 768 active threads per Compute Unit and ATI's GPU support 1024 active threads per Compute Unit. Because of the local size, our experiments use 256 work-items as one work-group which means only 33.3% and 25% capability of Compute Unit be used in NVIDIA's and ATI's GPU separately. Therefore Programmers should organize as much as possible work-items into work-group to fully employ the compute resource of device.

II-B. Use shared memory

Because of on chip, the shared memory is much faster than the device memory. In fact, accessing the shared memory is as faster as accessing a register, and the shared memory latency is roughly 100x lower than the device memory latency tested in our experiments. To achieve high memory bandwidth for concurrent accesses, the shared memory is divided into equally sized memory banks, which can be accessed simultaneously. However if multiple addresses of a memory request map to the same memory bank, the accesses are serialized and create banks conflict. To get maximum performance, the most important point is to use shared memory but avoid banks conflict. The local memory of OpenCL corresponds the shared memory on devices. In order to run faster and avoid re-fetching from device memory, programmers should put the data into the local memory ahead of complex and repeated computing.

II-C. Use vector data

In Intel's CPU and Nvidia'GPU, there are special vector instruction set for vector data. With vector data types, each work-item can processes N elements which benefit from hand-tuned vectorization of kernel code. In our kernel code, we adopt uchar16 instead of uchar to unroll the for loop in the procedure of SAD computing. Fig.**??** and Fig.**??** show the example code of no vectorization and vectorization separately. The vectorization of kernel code has 50%-80% performance enhancement which has been verified in our experiments.

III. PARALLELIZED MOTION ESTIMATION ALGORITHM

About motion estimation, there are two aspects considered to distribute some workloads to GPUs or CPUs, massively parallelized motion search and calculation of the matching criterion.

```
for(int m = 0; m < 16; m ++) {
   for(int n = 0; n < 16; n ++) {
    cost += abs_diff(current_mb[m + n], ref_mb[m + n]);
   }
}
```

Fig. 1. No vectorization code of SAD calculation

```
for(int m = 0; m < 16; m ++) {
   uchar16 cur_16 = (current_mb[m], current_mb[1 + m], current_mb[2 + m ], current_mb[3 + m],
   ......
   current_mb[12 + m], current_mb[13 + m], current_mb[14 + m], current_mb[15 + m]);

   uchar16 ref_16 = (ref_mb[m], ref_mb[m+ 1], ref_mb[m + 2], ref_mb[m + 3],
   ......
   ref_mb[m + 12], ref_mb[m + 13], ref_mb[m + 14], ref_mb[m + 15]);

   cost += abs_diff(cur_16.s0, ref_16.s0);cost += abs_diff(cur_16.s1, ref_16.s1);
   cost += abs_diff(cur_16.s2, ref_16.s2);cost += abs_diff(cur_16.s3, ref_16.s3);
  ......
   cost += abs_diff(cur_16.se, ref_16.se);cost += abs_diff(cur_16.sf, ref_16.sf);
  }
```

Fig. 2. Vectorization code of SAD calculation

We chose the Sum Absolute Difference (SAD) as the matching criterion for selecting the best Motion Vectors (MV).

$$SAD = \sum |windows(i) - image(i)|^2 \tag{1}$$

post code of SAD here with transferm of HMPP Because of the abounding workload of nested *for* loops, the classic full search ME algorithm cannot satisfy the demands of the real-time applications. As shown in Fig.**??**, our proposed method apply two OpenCL kernels pipelines which can be executed in parallel to replace the classic nested loops for SADs computing and comparing separately.

In the host, we pad the image when we read these frames from source file. Suppose that we have a $width \times height$ image, and a w size search window . The padded image size should be $(width + 2 \times w) \times (height + 2 \times w)$ as shown in Fig.**??**. Padding image guarantees no memory accessing violation when window moves on boundary of image and avoid performance decreasing caused by if-else statement in OpenCL kernel. Then we transfer the needed frames of test sequence at one time to the devices(CPU or GPU) memory to make use of the large memory bandwidth.

In all, we divide the workload of the proposed algorithm into two OpenCL kernels. One is in charge of computing the SADs , the other one is in charge of comparing these SADs to select the best (final MV). We define the two OpenCL kernels as *kernel_compute* and *kernel_compare*.

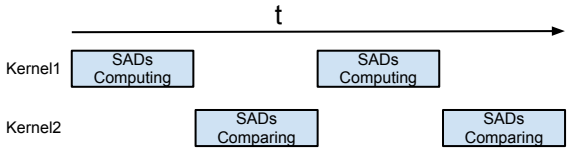

Fig. 3. Kernel pipelines of parallelized ME

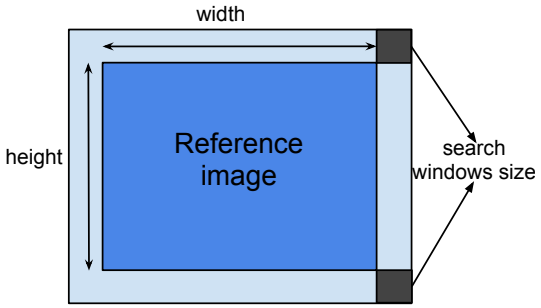

Fig. 4. Padding image

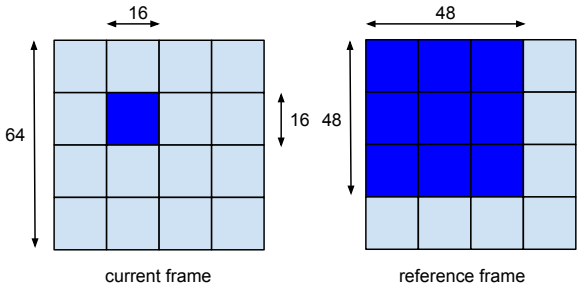

Fig. 5. Proposed calculation of SADs in local memory

III-A. SAD computation

Based on the block marching method, every frame is divided into macroblocks (MBs) ($MB\_size = MB\_width \times MB\_height = 16 \times 16 = 256$) which have a search region in the reference frame. We suppose the $Search\_Range = 32$. So every MB has $32 \times 32 = 1024$ candidates which concern $(48) \times (48) = 2304$ pixels in the reference frames as shown in the Fig.**??**.

In our design, we set the $localsize = MB\_size$ in NDRange and there are 256 work-items executed in one work-group ($work\_group\_size = 256$). Because each work-item process four candidates SAD, 1024 candidates SAD of one MB are distributed to one work-groups perfectly. For every frame, the total number of work-groups is N:

$$N = \frac{width}{MB\_width} \times \frac{height}{MB\_height} \times \frac{Search\_Range^2}{work\_group\_size} \quad (2)$$

When an OpenCL program invokes a kernel, N work-groups are enumerated and distributed as thread blocks to the multiprocessors with available Compute Units of CPU or GPU.

In our kernel *kernel_compute*, all the pixels of MB are transferred into local memory ($local[256]$) by the 256 work-items in one work-group. Until all the work-items in the same work-group reach the synchronous point using *barrier()* function, all the 256 work-items continue transferring the 1024 candidates (2304 pixels of search region concerned) of reference image into local memory ($local\_ref[2304]$). This differentiates our approach from approaches of [**?**]and[**?**] accelerated the full search motion estimation in H.264 with CUDA. In their work, the current MB is stored in local memory, but the search region of the reference frame is still in the global memory which results in inevitable re-fetching from global memory.

At the end, we adopt full search strategies to calculate the 1024 candidate SADs in local memory without re-fetching from the global memory. The amount of memory access is significantly reduced for better performance according to our experiment results. After calculation, all the 1024 candidate SADs are stored back to global memory ($cost[1024]$).

III-B. SAD comparison

In our kernel *kernel_compare*, we select the best candidate from $cost[1024]$ in three steps with 256 work-items as shown in Fig.**??**. First, the $cost[1024]$ is put into local memory. Second, each work-item find the minimum from 4 candidates as equation**??**.

$$\begin{aligned} cost[i] = Min(Min(cost[i], cost[i + strid]), \\ Min(cost[i + 2 \times strid], cost[i + 3 \times strid])) \end{aligned} \quad (3)$$

(i is the index of work-items, $i \subseteq [0, 255], strid = 256$). Last, we use parallel reduction method [**?**] which adopt x times iterations ($2^x = 256, x = 8$) to find the smallest candidate (final MV) from the remanent 256 candidates as shown in Fig.**??**.

III-C. Experiments result

To evaluate the performance of our proposed ME algorithm with OpenCL, we test in three different platforms which support OpenCL 1.1: Intel I7 2630qm(2.8Ghz), NVIDIA Geforce GT540m and AMD Radeon HD 6870. First, we run the reference C code of full search ME with single CPU core. Then, we run the proposed ME algorithm on the OpenCL CPU platform and two different GPUs platforms. We chose three test sequences with different resolution, Foreman (CIF, 352x288), City (4CIF, 704x576), Mobal_ter (720p, 1280x720) and the same 32x32 search range. As shown in Fig.**??**, for the different resolutions from CIF to 720P, our proposed method with OpenCL CPU achieves (107,19,7.6), with GT540m achieves (355,88,38) fps and with ATI HD6870 achieves (775,200,89) fps compared with C code that only produces (7.5,1.7,0.6) fps. So

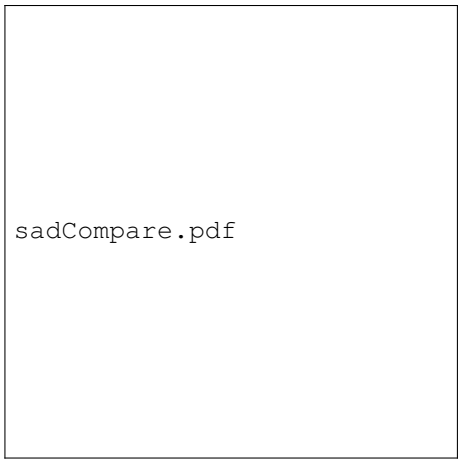

Fig. 6. SADs comparison in parallel

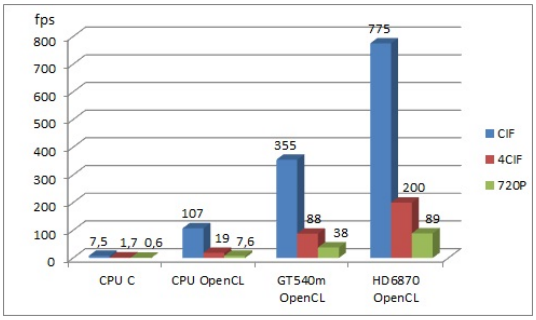

Fig. 7. Performance comparison with different platforms

our proposed method with AMD HD6870 achieves 100x to 150x speed-up for different resolutions compared with implementation with single CPU core. The CPU I7 2630qm with 4 physical cores and 8 threads achieves more than 10 times speed-up due to the utilization of vectorization and the unroll procedure.

In the meantime, experiments results show that the performance of the proposed ME algorithm have a close relationship with the number of Compute Units. As shown in Fig.**??**, the HD 6870 have the twice performance compared with GT540m (HD 6870 has more than 10 Compute Units and GT540m has 2 Compute Units).

IV. HETEROGENEOUS PARALLEL COMPUTING WITH OPENCL

In this section, we adopt the special feature of OpenCL - Heterogeneous Parallel Computing[**?**] to dig for the better performance of our proposed algorithm. First we build one heterogeneous parallel computing environment with one CPU(I7 2630qm) and one GPU(GT540m) as coprocessors for our arithmetic data-parallel computing with OpenCL. Then we distribute the workload to CPU device and GPU device separately. In such a heterogeneous computing system, we should determine the usage model of multi-devices and workload distribution. In order to minimum the cost of memory transferring between different devices, We prefer to use "multi-input" and "multi-output" for multi devices separately. Under the perfect condition, every devices complete their's workload at the same time.

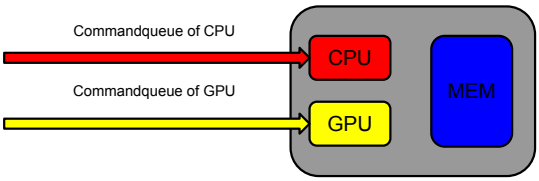

Fig. 8. Combined context for our heterogenous computing system

```
cl_uint num_devices;
cl_device_id devices[2];
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[0], &num_devices);
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[1], &num_devices);
context = clCreateContext(0, 2, devices, NULL, NULL, &error);
cl_command_queue commandQueue_cpu, commandQueue_gpu;
commandQueue_cpu = clCreateCommandQueue(context, devices[0], 0 , &error);
commandQueue_gpu = clCreateCommandQueue(context, devices[1], 0 , &error);
.....
```

Fig. 9. Example code of build combined OpenCL context for multi-devices

IV-A. Cooperative multi-devices usage model

For heterogeneous computing with different devices, we should build one cooperative multi-devices model. In OpenCL, there are two kind of models for multi-devices:

1) Separate contexts for different devices
2) One combined context for multi-devices

Because it is difficult to share the memory object and synchronies commandqueues of different contexts, so we prefer to build one combined context for CPU and GPU as shown in fig.**??**. The example code of building the combined context for CPU and GPU is shown in the fig.**??**.

IV-B. Workload distribution

With fixed problem size, how to distribute the workload to different devices is the key point to gain the enhancement of performance in heterogeneous computing system. In the video applications, the basic problem unit is frame or image.

Suppose that, our test video sequences have N frames. We transfer M frames to CPU device and (N - M) frames to GPU device to execute our proposed ME algorithm ($kernel\_compute$ and $kernel\_compare$). The suitable number of M does guarantee the best performance. As discussed in section **??**, the time of transferring the

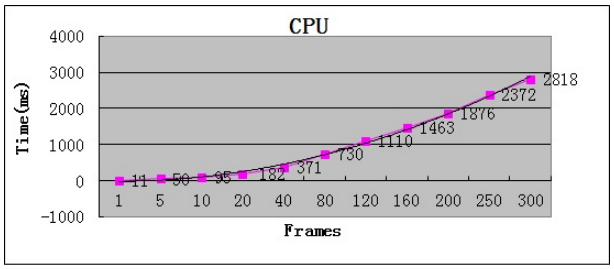

Fig. 10. Experimental curves of execution with CPU

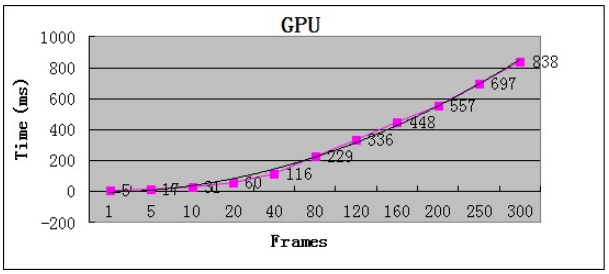

Fig. 11. Experimental curves of execution with GPU

needed frames is $T_{memtrans}$. The time of kernel computing is T_{kernel}. The time of copy buffer for kernel executed is $T_{memcopy}$. The total cost time of is

$$T_{device} = T_{memtrans} + T_{kernel} + T_{memcopy}. \quad (4)$$

Suppose that, $T_{cpu}(M)$ is the time of CPU processes M frames, and $T_{gpu}(N-M)$ is the time of GPU processes (N - M) frames. Under the perfect situation, the CPU and GPU will finish their work separately at the same time. When $T_{cpu}(M) = T_{gpu}(N-M)$, we can get the best performance of heterogeneous computing system.

IV-C. Find the balance of performance

At first, we should take some experimental curves to obtain the expressions of $T_{cpu}(M)$ and $T_{gpu}(N-M)$ as shown in the fig.**??** and the fig.**??**. As shown in the equation **??**, the total cost time which contains $T_{memtrans}$ is not be linear. So we can get the expressions of $T_{cpu}(M)$ and $T_{gpu}(N-M)$ from our experimental results with the technology of curve fitting.

$$T_{cpu}(x) = 29x^2 - 52x + 6 \quad (5)$$

$$T_{gpu}(x) = 8x^2 - 11x - 2.7 \quad (6)$$

According to equation **??** and equation **??**, we can calculate the suitable M as follow:

$$\begin{aligned} T_{cpu}(M) &= T_{gpu}(N-M) \\ 29M^2 - 52M + 6 &= 8(N-M)^2 - 11(N-M) - 2.7 \end{aligned} \quad (7)$$

Our test video sequence have 300 frames ($N = 300$), so the final suitable M is 105 which is calculated from equation **??**. When $M = 105$, we measure the performance of proposed method based on the heterogeneous computing system as shown in the Fig.**??**. The experimental results illustrate the obviously enhancement of performance.

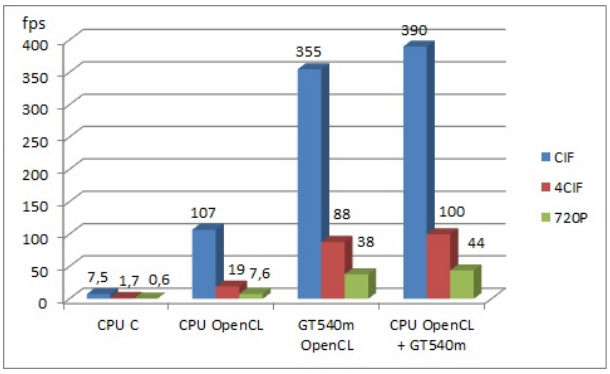

Fig. 12. Performance comparison with heterogeneous parallel computing

V. CONCLUSION

OpenCL provides one simple way to build heterogenous computing system and opportunities to increase the performance of parallel application. In this paper, we have presented one parallelized ME algorithm with OpenCL and some optimization strategies applied in our method. We test the proposed algorithm on heterogeneous parallel system which contains CPU and GPU. We also have developed one basic method to find the balance of workload on heterogeneous parallel computing system with OpenCL. Experimental results show that our method provides 100× to 150× speed-up compared with single CPU core implementation. Additionally we presents experimental results to show that we find the accurate method to distribute the workload in video applications based on heterogenous computing system which achieve obviously enhancement of performance.

So our future work is to design a automatic performance tuning OpenCL kernel code generator based on Open RVC-CAL Compiler (Orcc)[**?**] towards performance portability for heterogeneous parallel computing system. We aim to use the kernel code generator to generate the suitable parameters like localsize of kernel and determine the workload for different combination of devices automatically to get better performance for General Purpose date-parallel arithmetic computing with OpenCL.